CSCI 468 : Compilers

Team Member 1: Gaberial Darity

Team Member 2: Skylar Smoker

Instructor: Mr. Carson Gross

Spring 2021

Author Note

Section 1: Program

Section 2: Teamwork

Section 3: Design Pattern

Section 4: Technical Writing

Section 5: UML

Section 6: Design Trade-offs

Section 7: Software Life-cycle

Section 1: Program

Program is included in the source.zip file in this directory

Section 2: Teamwork

For CSCI 468 I have been working with Team Member 2 since the very beginning. We each developed our own parser. This development of our compilers was tests focused. We helped each other pass tests that we were struggling on. We often would work together on implementing some of the more difficult parts of the compiler. I created the documentation for Team Member 2's compiler while they made mine. This also allowed us to dive deeper into the code our partner had implemented. We could then see the others' thought process more clearly. Doing so allowed me to see areas where I needed to optimize my own code. We also created tests for each other. The tests I made for Team Member 2 utilized statements. This is so that my tests would reach more features in the compiler. Team Member 2 followed a similar vein of thought when sending me tests. The tests described are those sent to me. Test one tests the proper implementation of the execution methods for the function, if, print statement and the comparison expression. Test two shows proper implementation for the for loop with an if statement within it. Lastly test three tests the compile methods of the list literal expression. The list literal expression contained factor expressions within it.

Total Project Estimated Length: 200

Team Member 1:

Contributions: Primary Programmer, Tester, Tech Writer

Estimated Time Spent: 180

Team Member 2:

Contributions: Programming Assistance, Tester, Tech Writer

Estimated Time Spent: 20

Section 3: Design Pattern

Memoization Pattern:

The memoization pattern is used to make sure that a method does not execute more than once for the same inputs by storing the results into a data structure. The data structure is usually a Hashtable, HashMap, or an Array. We used this pattern with our CatscriptType getListType method. We used a HashMap as our data structure. The getListType method takes in a CatscriptType and outputs a ListType object. Without Memoization the ListType object would be created multiple times. Now, whenever there is a call to getListType we check the Hashmap. If the <CatscriptType, ListType> is not within the static HashMap we create a new instance of ListType. We then store that instance with the sent in CatscriptType. Lastly we return the ListType object. Now that it is stored if the same CatscriptType is sent in to getListType we do not need to create a new ListType.

Section 4: Technical Writing

Introduction

CatScript is a small programming language that is statically typed and is similar to the Java programming language. The language is considered a "toy" language but has a lot of interesting features, such as List Literals and Local Variable Type Inference. CatScript is based in Java and includes expressions, statements, a type system, and function declarations.

Features

For loops

The "for loop" consists of three variables: the identifier a.k.a. the variable, the expression, and the statement. The expression is contained inside the parenthesis next to the initial declaration "for" and the statement is contained within the body of the loop inside the brackets. A basic example of a "for loop" can be found below.

```
for (i in [3, 5, 7, 8]) {
    print(i)
}
```

The loop runs through each item in the list and prints put the value of i for each iteration. The loops should iterate 4 times because there is only 4 items in the given list.

If Statement

The if statement is made up of an expression, a statement, and more statements may be added on optionally. If there are additional statements, and the condition in the expression is not met, then the if statement will return null and continue on to the rest of the code. Below you can find an example of an if statement.

```
if (x > 5) {
    x = x + 1
}
```

The if statement checks if the expression evaluates to True and if so it will evaluate the statement and then continue on to the rest of the code.

```
if (x > 5) {
    x = x + 1
} else {
    print(x)
}
```

In this example, we give a second statement to be evaluated in the event the initial expression is evaluated to False. If this is the case then the code would print out the value of x and then move on.

Print Statement

The print statement is made up of the print() declaration and an expression. The following example has a string as the expression and prints the word "Hello."

```
print("Hello")
```

Variable Statement

The variable statement is made up of an identifier, an optional type expression, and an expression. The optional type expression is used to keep track of type inference. During the execution of the program these are used to place data onto the scope of the program.

```
var x : int = 5
```

Assignment Statement

The assignment statement is made up of an identifier and an expression. The identifier must be unique in the current scope.

```
var y = 20
y = 15
```

The statement is validated by making sure the identifier is unique and that the expression is of a compatible type.

Return Statement

The return statement contains the word "return" followed by an expression. Each return statement must be placed within a function with a defined return type.

```
function int TestFunction(x : int) {

   var z = x + x
   return z
}
```

When the method is evaluated, a return exception is thrown to allow for the program to be notified that the function has had a return evaluated.

Function Statements

Function Declaration Statement

The function declaration statement consists of an identifier, parameter list, a type expression, and a function body statement. The identifier must be unique and each parameter needs a type expression to define the return type.

```
function void TestFunction(x) {
    print(x)
}

function void DeclaredTestFunction(x : int) {
    print(x)
}
```

Function Body Statement

The function body statement is contained inside a function declaration. The statement can contain any type of expression, including another function call. The body must include a return statement when a return type is declared.

```
function void TestFunction(x) {
    // This is the function body
    var x = 0
    x = x + 1
    print(x)
}
```

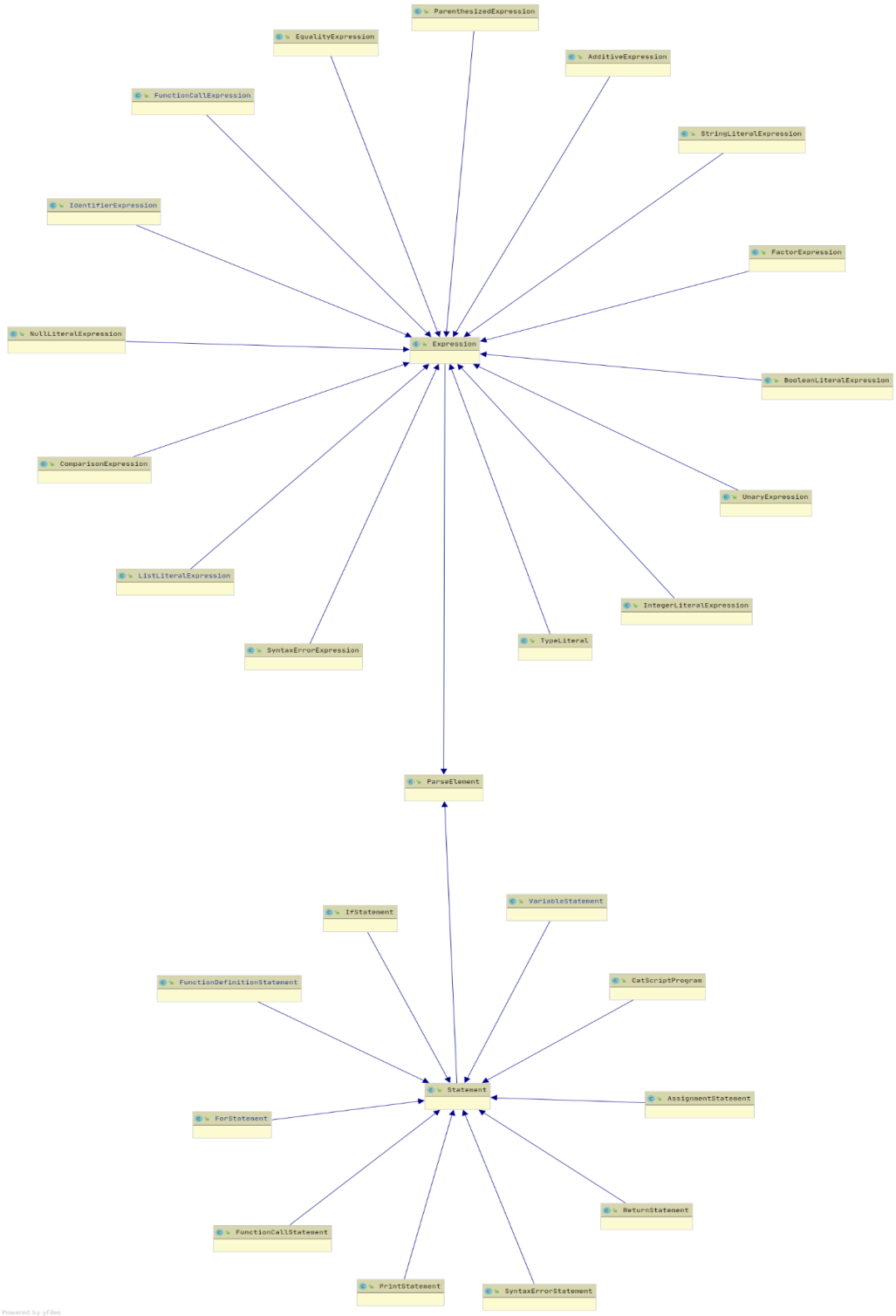
List Literal Expression

The list literal expression consists of a list of expressions. Each expression of the same list must be of the same type.

```
["This", "Is", "A", "List"]
[1, 2, 3, 4]
[False, True, True]
```

The compiler validates all items contained in the list and then performs type checking to ensure there are no type discrepancies. The final list is returned in the form of an ArrayList.

Section 5: UML



Section 6: Design Trade-offs

One of the main design trade-offs would be using recursive descent parsing rather than a parser generator. An amazing aspect of using recursive descent, was how closely it followed the grammar. The grammar was used as if it was pseudocode to help us implement our parser. Something that recursive descent parsing does poorly is error messages. It is difficult to provide good error messages. The error messages need to be contained and passed all the way out of the recursive calls. Recursive descent also has a lot of function calls and stack pushes. This takes up more space than when using a parser generator. Another design decision would be the use of if statements. I did not implement switch statements because I find that if statements add to the readability of the program. Switch statements do look nicer, but I find that they are hard to follow for some of the very long control statements we needed. Examples would be parsePrimaryExpression, parseFunctionDeclaration, and parseVariableStatement. Lastly, as talked about previously we used the memoization design pattern. This does come with a trade off. It will shorten the time complexity of the getListType function, but at a cost. Now we will be taking up even more space within our program. This is because of the static HashMap that will contain all of the called and created CatscriptType, ListType pairs. Another thing we decided to do was make our lists read only. You cannot add or modify lists that have been created in CatScript; they are immutable. This makes our lists covariant. You cannot cause a type error with lists after they are created. You also can’t add anything to them and that alone is a huge design trade off.

Section 7: Software Life-cycle

Test driven development was what we used in conjunction with this program. It has been an amazing experience. It really allows you to understand exactly what is expected of modular portions of your program. Test driven development allows for quick confirmation on what your code is doing and if it is doing it right. This is really useful, especially if you are working with someone else. They could come in and write code and have a better understanding of the logic behind the parts they are working on. A collaborator could also run all of the tests and see if their changes have caused failures in different parts of the program. Some downfalls of test driven development, when working on this program, is that it is limited. There are some areas of the code that you are unable to test or test easily. The methods within the statements and expressions that compile to bytecode are one such area. At times the tests would give a nondescript error and would not show the bytecode that was produced so far when failed. When working with test driven development on a larger project, writing tests could be something that bogs down a workflow. Especially if the group does not create quality tests. It is also worth mentioning that it could be something that overtakes actual programming in importance if you are not careful. The tests are created to help guide coding. It should not be the main focus, regardless of it being a useful tool.

OUTLINE

- Section 4: Technical Writing - Include the documentation generated by your partner for the catscript programming language
    - Need to paste the docs my partner writes for me